VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY
INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

DATA STRUCTURE AND ALGORITHMS

FINAL PROGRESS ASSESSMENT

Course by Dr. Trần Thanh Tùng

TOPIC: DABA 2048 GAME

TEAM MEMBERS

ITITIU20184	Phạm Đức Đạt
ITITIU20364	Huỳnh Lam Đạt
ITITWE20026	Phạm Vũ Bão

GitHub repository: https://github.com/ducdatit2002/daba-2048/
Deploy: https://ducdatit2002.github.io/daba-2048/

TABLE OF CONTENTS

TABLES OF FIGURES

ABSTRACT

The creation of a digital version of the well-known puzzle game 2048 is described in this report. In order to get to tile number 2048, players must slide numbered tiles onto a 4x4 grid in order to combine them into larger numbers. Our implementation keeps the core elements of the original game while incorporating new features that improve replay ability and user engagement. The robust JavaScript Game Manager class controls all the essential game mechanics, such as tile movement and combinations. On a range of devices, the matching JavaScript driver and cascading style sheet graphical user interface (GUI) offer a responsive and user-friendly experience.

The Model-View-Controller (MVC) design pattern is closely adhered to in this project. The MVC architecture is adapted to improve the code base's scalability and maintainability while streamlining development. Git version control enables thorough tracking of the development process, guaranteeing that collaborative work is synchronized and that a project's evolution is well documented.

To give players a more accommodating and customized gaming experience, new features have been added, like the capacity to reverse actions and save game progress. The report includes information on the game's rules, the software's organization, an example of gameplay, the application of design patterns, a thorough explanation of the graphical user interface, and the use of data structures and algorithms. The result demonstrates how contemporary web technologies and software design techniques can recreate and modernize classic games for the current gaming environment.

I. INTRODUCTION

1. ***Rules of game 2048*** [1]***:***

- Start playing: The game is played on a 4x4 square grid with any even number of squares. The first two value cells are filled with 2 or 4, the remaining cells are empty.

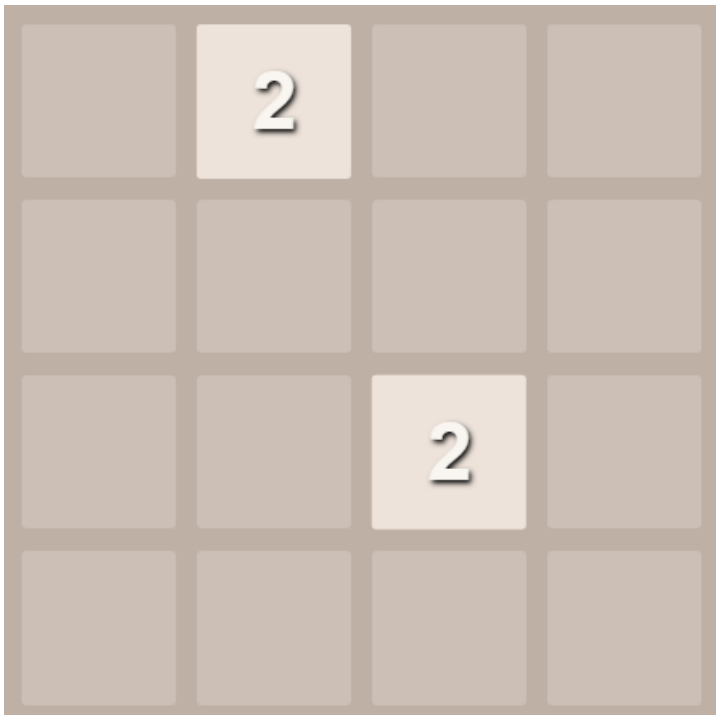

Figure 1 Start of game.

- Moving: Each turn, the player can move up, down, left, or right by pressing any one of the four keys. If a particular move modifies the grid, it is permissible to make that move. Any one of the four keys that we can press causes the cell's elements to move in that direction. If two similar numbers occur in that row or column, they add up, and the extreme cell in that direction fills with that number while the other cells empty again and any randomly located empty cell fills with two. Different value squares will have different colors for simple identification.

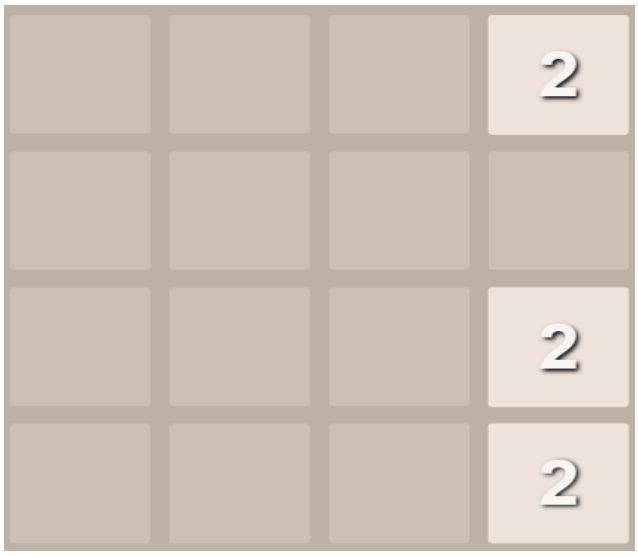

Figure 2 Before moving.

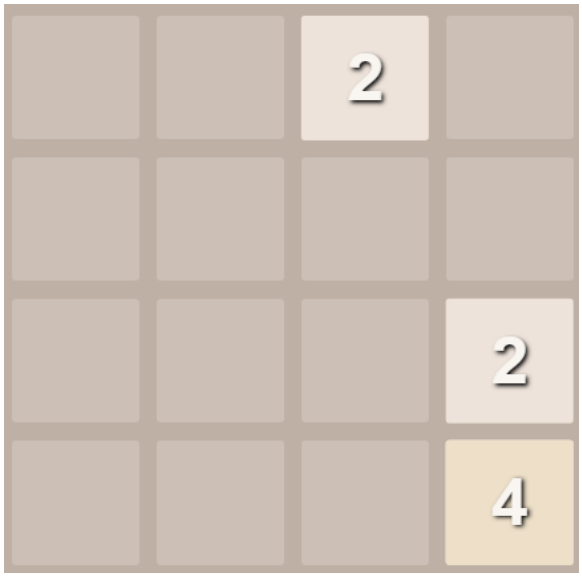

Figure 3 After moving.

- End of the game: The player wins when they produce a square with the number 2048. At this point, the player can opt to keep playing in order to achieve values greater than 2048. The game ends when there are no more valid movements (no empty cells and neighboring cells with different values).

Figure 4 Win the game (Ref: Internet).

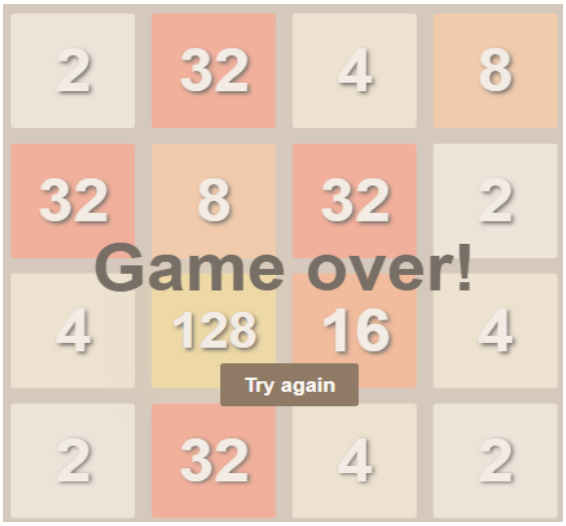

Figure 5 Game Over

2. ***Game scoring method:***

- Players earn points by matching number tiles. Every time two number tiles are combined; the player earns points equal to the total value of the newly created tile.
- The current score and highest score are displayed on the screen, allowing players to track their progress.

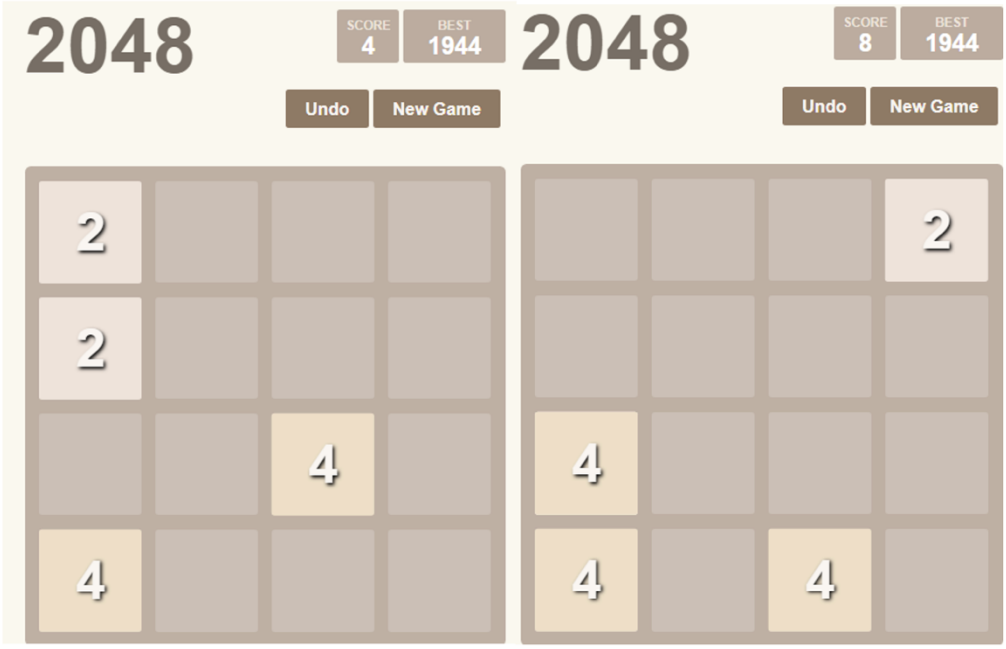

Figure 6 Differences after combined.

3. Undo function:

- Purpose of Undo: The undo function allows players to undo their last action, as a means of correcting errors or changing strategies.
- How to Use Undo:

When a player makes a move and wants to return to the previous state, they can press the undo button to undo that move.

The game will return to the state just before the last move was made, and the score will be adjusted accordingly.

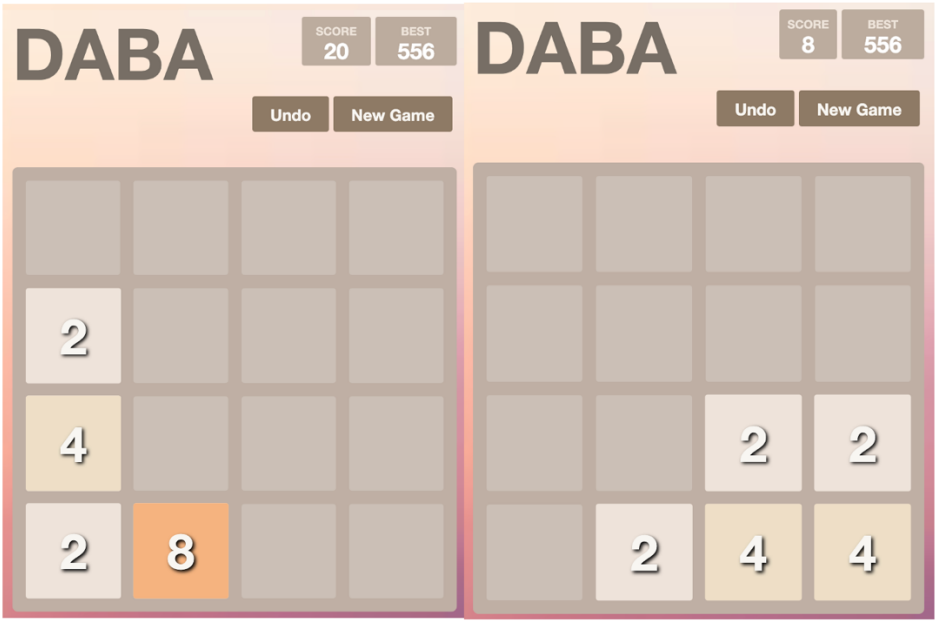

Figure 7 Differences of game after using Undo Function.

II. CLASS DIAGRAM

Figure 8 Class diagram.

1. The **GameManager** class represents the main game logic for a game similar to "2048". Here's a brief description of its functionality:

   - **Constructor**: Initializes the game by setting the grid size, input manager, actuator, storage manager, and the number of starting tiles. It also sets up event listeners for user input events like move, restart, undo, and keep playing.

   - restart(): Restarts the game by clearing the last moves, clearing the game state, and resetting the game's properties. It also clears the game won/lost message and calls the setup() method.

   - undo(): Reverts the last move by retrieving the previous game state from storage, setting the game state to the previous state, and continuing the game. It also clears the game won/lost message and calls the setup() method.

   - keepPlaying(): Sets the keepPlaying property to true, allowing the user to keep playing after winning the game. It clears the game won/lost message.

- isGameTerminated(): Checks if the game is over (lost) or won and the user hasn't chosen to keep playing. Returns true if the game is terminated; otherwise, returns false.

- setup(): Sets up the game by either reloading the previous game state or creating a new game state. It creates a grid, initializes the score, over, won, and keepPlaying properties, and adds the initial tiles. Finally, it updates the game's UI.

- addStartTiles(): Adds the specified number of starting tiles to the grid.

- addRandomTile(): Adds a new tile (with a value of 2 or 4) to a random available position on the grid.

- actuate(): Sends the updated grid and game information to the actuator for rendering. It also updates the best score in the storage manager, saves or clears the game state based on whether the game is over or not, and enables or disables the undo button based on the availability of the last move.

- serialize(): Represents the current game state as an object, including the grid, score, over, won, and keepPlaying properties.

- prepareTiles(): Saves the current tile positions and removes merger information by iterating over each cell in the grid.

- moveTile(): Moves a tile from its current position to a new position on the grid.

- move(): Moves the tiles on the grid in the specified direction (up, right, down, or left). It prepares the tiles for movement, traverses the grid in the specified direction, merges tiles if possible, moves the tiles to their new positions, updates the score, checks for winning condition (2048 tile), adds a random tile, checks for game over condition, and updates the game's UI.

- getVector(): Returns the vector representing the chosen direction (up, right, down, or left).

- buildTraversals(): Builds a list of positions to traverse in the right order based on the chosen direction.

- **findFarthestPosition():** Finds the farthest position in the grid in the specified direction where a tile can be moved.

- movesAvailable(): Checks if there are any moves available by checking if there are any empty cells or if there are any matching adjacent tiles.

- **tileMatchesAvailable():** Checks if there are any available matches between tiles on the grid.

- **positionsEqual():** Checks if two positions (cells) on the grid are equal.

2. The **KeyboardInputManager** class is responsible for managing user input events related to keyboard and touch controls in a game. It provides methods for registering event handlers, emitting events, and listening for user input. Here is a brief description of the class and its methods:

   - **KeyboardInputManager**: The constructor function initializes the events object to store event handlers. It determines the appropriate touch events based on the browser being used and calls the listen method to start listening for user input.

   - **on(event, callback)**: Registers an event handler for the specified event. If the event does not exist in the events object, a new array is created to store the callbacks.

   - **emit(event, data)**: Triggers the specified event by calling all the registered callbacks associated with that event. The data parameter is passed to the callbacks.

- **listen()**: Sets up event listeners for keyboard keys, button presses, and swipe events. It maps keyboard keys to game actions and emits corresponding "move," "restart," "keepPlaying," and "undo" events based on user input.
- **restart(event)**: Event handler for the restart action triggered by keyboard or button press. Emits the "restart" event.
- **keepPlaying(event)**: Event handler for the keep playing action triggered by a button press. Emits the "keepPlaying" event.
- **bindButtonPress(selector, fn)**: Binds a button press event to the specified selector. When the button is clicked or touched, it calls the provided function **fn** and binds the **this** context to the KeyboardInputManager instance.
- **undo(event):** Event handler for the undo action triggered by a button press. Emits the "undo" event.
- **window.fakeStorage**: A fake implementation of the localStorage API used for storing game data. It provides methods for setting, getting, removing, and clearing data stored in memory.

3. The **Grid** class represents a grid structure used in a game. It provides methods for managing the grid's cells, inserting and removing tiles, checking availability of cells, and serializing the grid's state. Here is a brief description of the class and its methods:
- **Grid(size, previousState)**: The constructor function initializes a grid of the specified size. If a previousState is provided, it creates the grid based on the state. Otherwise, it creates an empty grid.
- **empty()**: Creates and returns an empty grid by initializing a 2D cells array filled with null values.
- **fromState(state):** Creates and returns a grid based on the provided state object. It iterates over the state and creates tiles for non-null values.
- **randomAvailableCell():** Finds and returns the first available random position (cell) in the grid. It uses the availableCells() method to determine the available cells and selects a random one.
- **availableCells():** Returns an array of available cells in the grid. It iterates over each cell using the eachCell() method and adds the coordinates of cells without tiles to the cells array.
- **eachCell(callback):** Calls the provided callback function for every cell in the grid. It iterates over each cell using nested loops and passes the cell's coordinates and the tile (if present) to the callback.
- **cellsAvailable():** Checks if there are any available cells in the grid by checking if the availableCells() array has a length greater than zero.
- **cellAvailable(cell):** Checks if the specified cell is available (not occupied by a tile) in the grid by calling cellOccupied().
- **cellOccupied(cell):** Checks if the specified cell is occupied (contains a tile) in the grid by checking if the corresponding cell content is truthy.
- **cellContent(cell):** Returns the content (tile) of the specified cell in the grid. If the cell is within the grid boundaries, it returns the corresponding cell content; otherwise, it returns null.
- **insertTile(tile):** Inserts the specified tile into the grid at its position by updating the corresponding cell in the cells array.
- **removeTile(tile):** Removes the specified tile from the grid by setting the corresponding cell in the cells array to null.

- **withinBounds(position):** Checks if the position is within the boundaries of the grid. It returns true if the position has x and y coordinates within the range of 0 to size - 1, where size is the size of the grid.

- **serialize():** Converts the grid and its contents into a serializable object. It iterates over each cell in the grid, calling the serialize() method on each tile (if present) to obtain its serialized representation. It returns an object containing the size of the grid and the serialized cells array.

4. The **HTMLActuator** class is responsible for updating the HTML view of a game based on the state of the grid and metadata. It interacts with various elements in the HTML document to display the game tiles, score, messages, and other visual elements. Here is a brief description of the class and its methods:

- **HTMLActuator()**: The constructor function initializes the HTMLActuator object. It selects HTML elements from the document such as the tile container, score container, best score container, and message container. It also initializes the score property to zero.

- **actuate(grid, metadata)**: The actuate method updates the HTML view based on the provided grid and metadata. It clears the tile container, then iterates through each cell of the grid and adds the corresponding tile to the HTML view using the addTile method. It also updates the score and best score using the updateScore and updateBestScore methods, respectively. If the game is terminated, it displays the appropriate message based on the over and won properties of the metadata.

- **continueGame():** The continueGame method clears the game message, allowing the player to continue playing the game.

- **clearContainer(container):** The clearContainer method removes all child elements from the specified container element.

- **addTile(tile):** The addTile method adds a tile to the HTML view. It creates the necessary HTML elements to represent the tile and applies appropriate classes based on its value and position. If the tile has a previous position or has merged from other tiles, it updates the position or renders the merged tiles first. Finally, it appends the tile to the tile container in the HTML view.

- **applyClasses(element, classes):** The applyClasses method sets the CSS classes of the specified element to the provided classes array.

- n**ormalizePosition(position):** The normalizePosition method adjusts the position coordinates by adding 1 to each coordinate. This is used to generate the CSS class representing the position of a tile.

- **positionClass(position):** The positionClass method generates a CSS class name based on the normalized position coordinates.

- **updateScore(score):** The updateScore method updates the displayed score in the HTML view. It calculates the difference between the new score and the previous score, updates the score property, and displays the new score in the score container. If the difference is positive, it adds a score addition element to indicate the increase.

- **updateBestScore(bestScore):** The updateBestScore method updates the displayed best score in the HTML view.

- **message(won):** The message method displays a game message based on whether the player won or lost. It adds the appropriate class to the message container and sets the message text accordingly.

- **clearMessage():** The clearMessage method removes the game message by removing the game-won and game-over classes from the message container.

5. The **LocalStorageManager** class is responsible for managing game data in the browser's local storage. It provides methods to store and retrieve data related to the game's best score, game state, and moves history. Here is a brief description of the class and its methods:
   - **LocalStorageManager()**: The constructor function initializes the LocalStorageManager object. It sets the keys for storing the best score, game state, total moves, and last moves in the local storage. It also determines whether the browser supports local storage and assigns the appropriate storage object (window.localStorage or window.fakeStorage).
   - **localStorageSupported()**: The localStorageSupported method checks if the browser supports local storage by attempting to set and remove an item from the storage. It returns true if the operations are successful, indicating that local storage is supported, and false otherwise.
   - **getBestScore()**: The getBestScore method retrieves the best score from the local storage. If no best score is found, it returns 0.
   - **setBestScore(score)**: The setBestScore method stores the given score as the new best score in the local storage.
   - **getGameState()**: The getGameState method retrieves the game state from the local storage. It parses the stored JSON string and returns the corresponding JavaScript object. If no game state is found, it returns null.
   - **setGameState(gameState)**: The setGameState method stores the provided gameState object as a JSON string in the local storage.
   - **clearGameState()**: The clearGameState method removes the game state from the local storage.
   - **getLastMove(willUse)**: The getLastMove method retrieves the last move from the local storage. It takes a boolean parameter willUse indicating whether the move will be used. If willUse is true, it removes the last move from the storage and updates the total moves count accordingly. It returns the parsed JSON object of the last move. If no last move is found, it returns null.
   - **setLastMove(lastMove)**: The setLastMove method stores the provided lastMove object as a JSON string in the local storage. It also increments the total moves count and updates it in the storage.
   - **clearLastMoves()**: The clearLastMoves method removes all the stored last moves and the total moves count from the local storage.
   - **getTotalMoves()**: The getTotalMoves method retrieves the total moves count from the local storage. If no count is found, it returns 0.
   - **setTotalMoves(moves)**: The setTotalMoves method stores the provided moves count in the local storage.
6. The **Tile** class represents a tile object in the game. Each tile has a position, a numeric value, and can potentially have a previous position and be merged from other tiles. Here is a brief description of the class and its methods:
   - **Tile(position, value)**: The constructor function initializes a Tile object with the provided position and value. If no value is provided, it defaults to 2. It also sets the previousPosition and mergedFrom properties to null.

- **savePosition()**: The savePosition method saves the current position of the tile by assigning it to the previousPosition property. This is useful for tracking tile movements.

- **updatePosition(position)**: The updatePosition method updates the position of the tile to the provided position. It assigns the x and y coordinates of the position to the corresponding properties of the tile.

- **serialize()**: The serialize method returns a serialized representation of the tile as a JavaScript object. It includes the position object with x and y coordinates, as well as the value of the tile.

III. GRAPHICAL USER INTERFACE

1. Animation

- Animations like "move-up" and "fade-in" enhance the user experience by adding visual feedback to actions like number tiles and game messages.

- New number tiles (tile-new) and merged number tiles (tile-merged) will display with a "pop" effect to highlight their appearance on the board.

2. User Interaction

- Buttons like "Restart" and "Undo" have bold backgrounds and prominent text colors, making them easy to spot and interact with.

- Elements like links (a) and important text (strong.important) are styled to attract players' attention.

3. *Mobile responsive*

- The CSS file also contains media queries to adapt the look and feel to devices with smaller screens such as mobile phones, ensuring the game can be played comfortably on all types of devices. different devices.

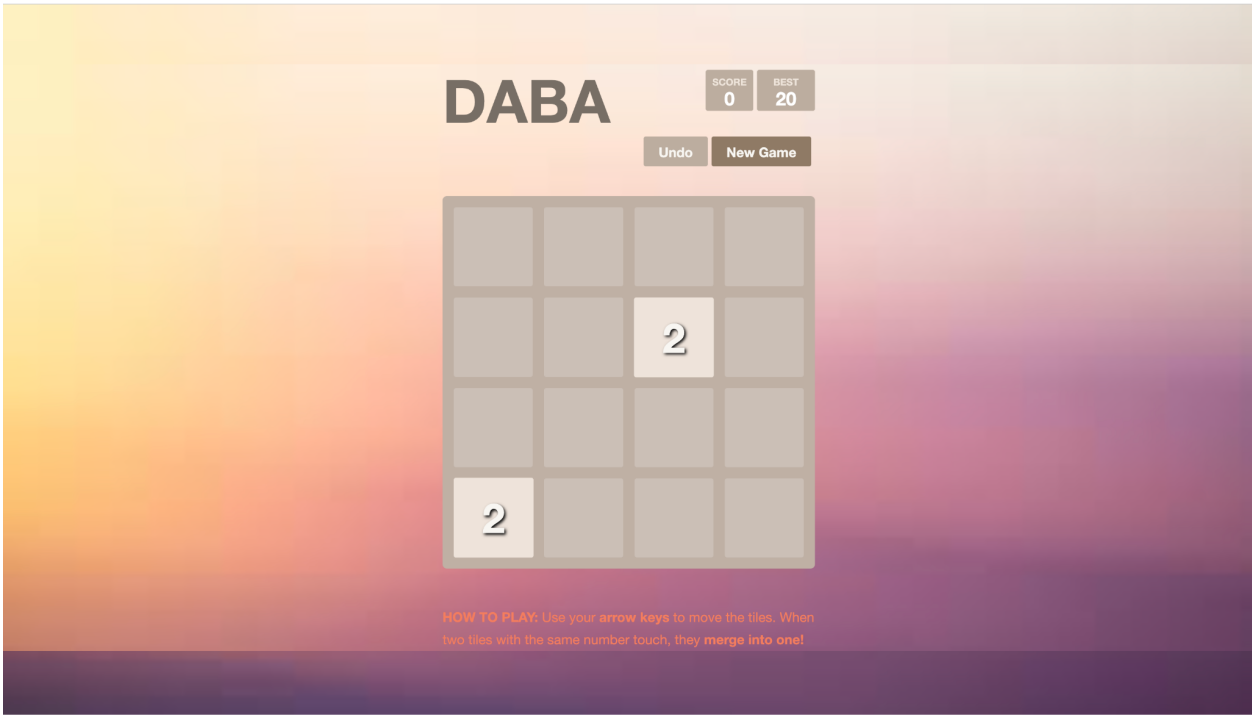

Figure 9 User interface in Desktop

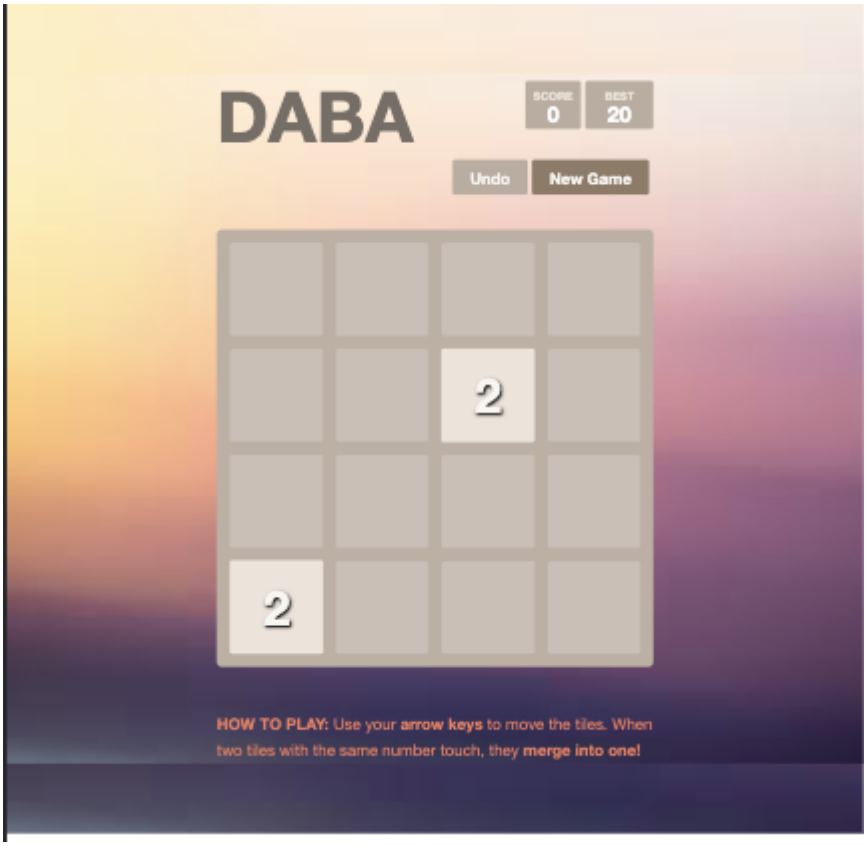

Figure 10 User interface in Mobile Phone

IV. APPLYING DESIGN PATTERN

1. MVC Model Concept

The MVC pattern is a software design model used to separate application logic into three main parts: Model, View, and Controller.

- **The model represents** the core data structure of the application, as well as the logic associated with that data. It directly manages the application's data, logic, and rules.
- **View represents** any output of information (UI), such as a drawing on a screen or a panel. In the case of 2048, this is where the game is displayed to the player.
- **The Controller processes** input from the user, converting it into requests for the Model or View.

2. Applying MVC Model in DABA 2048 Game

Model:

- Grid.js manages the state of the game grid and the elements within it.
- Tile.js defines and manages the state of each number tile in the game.
- LocalStorageManager.js handles storing and retrieving data from local storage, like high scores and game status.

View:

- index.html is the basic structure of the user interface.
- 2048.css styles the interface, making the game more attractive and easier to use.

Controllers:

- GameManager.js is where the main thread of the game is handled, combining the Model and View together.
- KeyboardInputManager.js handles input from the user and notifies GameManager about events.

- Windows.js

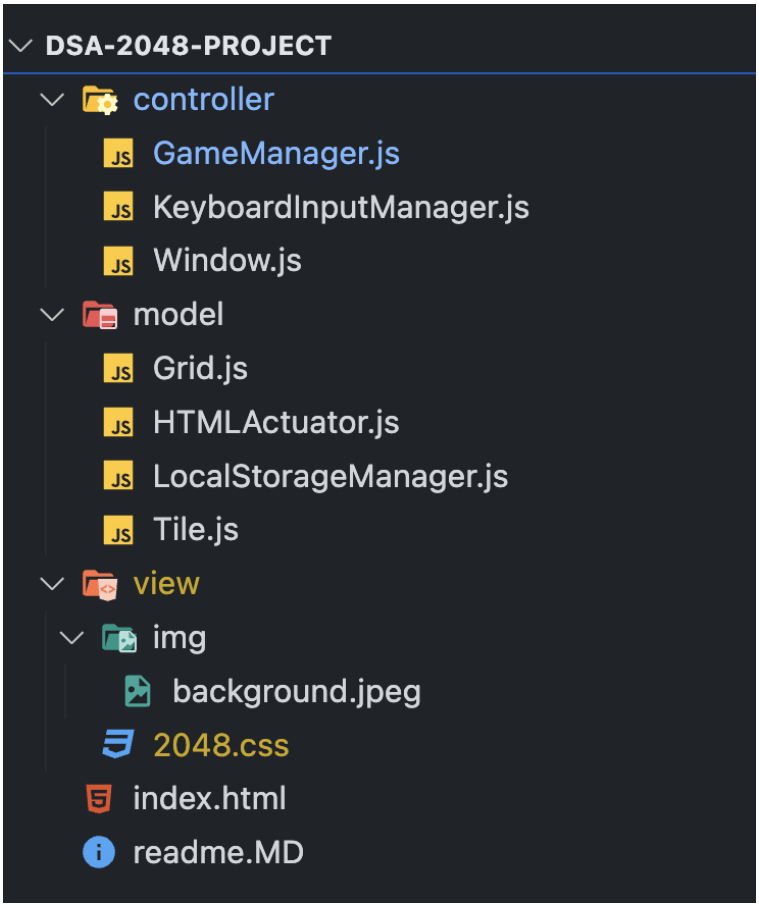

Figure 11 Software Architecture using MVC Design Pattern.

V. USING DATA STRUCTURE AND ALGORITHMS:

1. Arrays

Arrays in data structures and algorithms for managing gameplay, movement, and user interface elements. Here's how arrays are used in different elements of the game:

a. Grid Management:

- Initialization of Empty Grid: The two-dimensional array **Grid.prototype.empty** is used to create the game grid space. Empty cells are initialized with a **null** value, which allows identifying positions without numbered tiles.

```
// Build a grid of the specified size
Grid.prototype.empty = function () {
  var cells = [];

  for (var x = 0; x < this.size; x++) {
    var row = cells[x] = [];

    for (var y = 0; y < this.size; y++) {
      row.push(null);
    }
  }

  return cells;
};
```

Figure 12 Grid.prototype.empty method.

- Grid State Recovery: **Grid.prototype.fromState** uses a two-dimensional array to reconstruct the grid state from a saved game instance, allowing players to continue from the same state.

```
Grid.prototype.fromState = function (state) {
  var cells = [];

  for (var x = 0; x < this.size; x++) {
    var row = cells[x] = [];

    for (var y = 0; y < this.size; y++) {
      var tile = state[x][y];
      row.push(tile ? new Tile(tile.position, tile.value) : null);
    }
  }

  return cells;
};
```

Figure 13 Grid.prototype.fromState method.

2. Tile Management:

- Adding Starting and Random Tiles: The **GameManager.prototype.addStartTiles** and **GameManager.prototype.addRandomTiles** methods use arrays to manage the addition of new numbered tiles to the game grid, creating elements. randomization required for the game.

```
// Set up the initial tiles to start the game with
GameManager.prototype.addStartTiles = function () {
  for (var i = 0; i < this.startTiles; i++) {
    this.addRandomTile();
  }
};
```

Figure 14 GameManager.prototype.addStartTiles method.

```
// Adds a tile in a random position
GameManager.prototype.addRandomTile = function () {
  if (this.grid.cellsAvailable()) {
    var value = Math.random() < 0.9 ? 2 : 4;
    var tile = new Tile(this.grid.randomAvailableCell(), value);

    this.grid.insertTile(tile);
  }
};
```

Figure 15 GameManager.prototype.addRandomTiles method.

- Determining Empty Cells: **Grid.prototype.availableCells** is an array that stores empty positions in the grid, from which the algorithm can select an empty position to add a new cell to the grid.

```
Grid.prototype.availableCells = function () {
  var cells = [];

  this.eachCell(function (x, y, tile) {
    if (!tile) {
      cells.push({ x: x, y: y });
    }
  });

  return cells;
};
```

Figure 16 Grid.prototype.availableCells method.

- Movement Handling Algorithm: **GameManager.prototype.move** uses arrays to store move directions and implements move logic through **traversals.x** and **traversals.y** arrays, which determine the order in which cells will be placed. processed and moved on the grid.

```
// Move tiles on the grid in the specified direction
GameManager.prototype.move = function (direction) {
  // 0: up, 1: right, 2: down, 3: left
  var self = this;

  if (this.isGameTerminated()) return; // Don't do anything if the game's over

  var cell, tile;

  var vector     = this.getVector(direction);
  var traversals = this.buildTraversals(vector);
  var moved      = false;
```

Figure 17 GameManager.prototype.move method.

- State and Score Storage Management: **LocalStorageManager** uses arrays to store data such as highest scores and current game state, allowing for efficient game state saving and restoration.

```
function LocalStorageManager() {
    this.bestScoreKey     = "bestScore";
    this.gameStateKey     = "gameState";
    // Changes
    this.totalMovesKey    = "totalMoves";
    this.lastMoveKey      = "move#";

    var supported = this.localStorageSupported();
    this.storage = supported ? window.localStorage : window.fakeStorage;
  }
```

Figure 18 LocalStorageManager function.

2. *Stack*

Stack to perform the Undo function in the 2048 game. In this data structure, the last element added will be the first element removed (LIFO - Last In First Out).

Use Stack to Install Undo Function in Game 2048:

- The undo function allows the player to return to the previous state of the game. To do this, the game uses a stack to store a history of operations. When the player moves the numbered tiles on the board, the state before the move is "pushed" onto the stack. When the player chooses to undo, the last state in the stack (latest state) is "retrieved" to restore the previous game state.

A stack that stores game states is implemented as follows in the 2048 game source code:

- Save Status Before Moving: Use the serialize method to save the current state of the game to a variable.

```
// Changes
var dat = this.serialize();
```

Figure 19 Save Status Before Moving.

- Check If Any Move Is Performed:

If there is a move to be made (checked by the variable moved), the current state after the move is saved to the stack using the method **this.storageManager.setLastMove(dat);.**

```
// Changes
this.storageManager.setLastMove(dat);
```

Figure 20 Check If Any Move Is Performed.

- Undo Move:

When the undo function is called **(GameManager.prototype.undo = function ())**, method **this.storageManager.getLastMove(true)**; will retrieve the last state from the stack to restore the game state.

```
// Changes undo move
GameManager.prototype.undo = function () {
  var data = this.storageManager.getLastMove(true);
  if (data !== null) {
    this.storageManager.setGameState(data);
    this.actuator.continueGame();
    this.setup();
  }
};
```

Figure 21 Undo move.

VI. CONCLUSION

The creation and use of the JavaScript game 2048 is an intriguing blend of contemporary web technology and traditional game design. This project showcases the potential of modern web development tools and grand methods by not only recreating the well-known puzzle game but also improving it with cutting-edge features and a user-centered approach.

Key highlights of this project include:

1. Respect for MVC Architecture: The project guarantees a distinct division of responsibilities by closely following the Model-View-Controller pattern. This structure makes the code easier to update and manage by improving its extensibility and maintainability.
2. Improved user experience: Adding new features like the option to save game progress and undo actions greatly increases user engagement.

3. Responsive design: Paying attention to user interface responsiveness guarantees that games are playable and accessible on a variety of platforms, including desktop and mobile. This approach to universal design holds significance in the multi-device world of today.
4. Advanced Use of CSS and JavaScript: The project makes good use of CSS to create an aesthetically pleasing interface and JavaScript for game logic and controls. These technologies are used to demonstrate how powerfully they can be used to create dynamic and interactive web applications.
5. Effective data management is demonstrated by the way arrays and stacks are used to control game activity and state. These decisions improve user interaction and game performance.
6. Git-based collaborative development: Using Git for version control emphasizes how crucial collaboration tools are to contemporary software development. It guarantees that team members can collaborate easily and that there is thorough documentation of the project's progress.
7. Potential for future expansion: The architecture and design of the project make it simple to add new features and expand. Future additions like multiplayer modes, online leaderboards, and even different grid sizes and game variations could be made possible by this adaptability.

In conclusion, the JavaScript remake of the puzzle game 2048 is a fantastic illustration of contemporary web development techniques in addition to being a testament to the game's enduring appeal. This demonstrates how technology can bring back classic games, offering players and developers alike new challenges and educational opportunities.

Bibliography

[1] freeCodeCamp, "React Tutorial – How to Build the 2048 Game in React," 08 09 2021. [Online]. Available: https://www.freecodecamp.org/news/how-to-make-2048-game-in-react/#:~:text=2048%20Game%20Rules,%2C%2032%2C%20and%20so%20on..